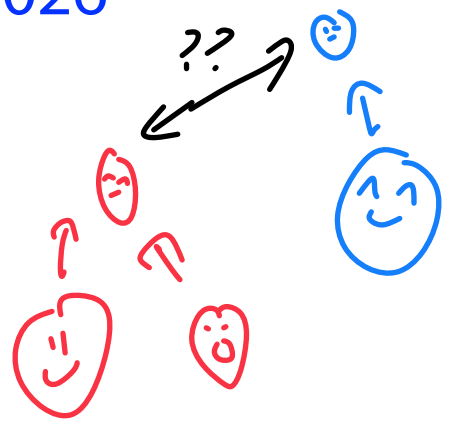
Data Structures

Disjoint Sets

CS 225

Brad Solomon

March 23, 2026



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Informal Early Feedback (Part 2)

This one is the largest of the feedback surveys

Includes questions about your lab sections

Exam 3 (3/23 — 3/25)

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam on PL

Topics covered can be found on website

Registration started March 5

https://courses.engr.illinois.edu/cs225/exams/

Learning Objectives

Introduce and implement disjoint sets

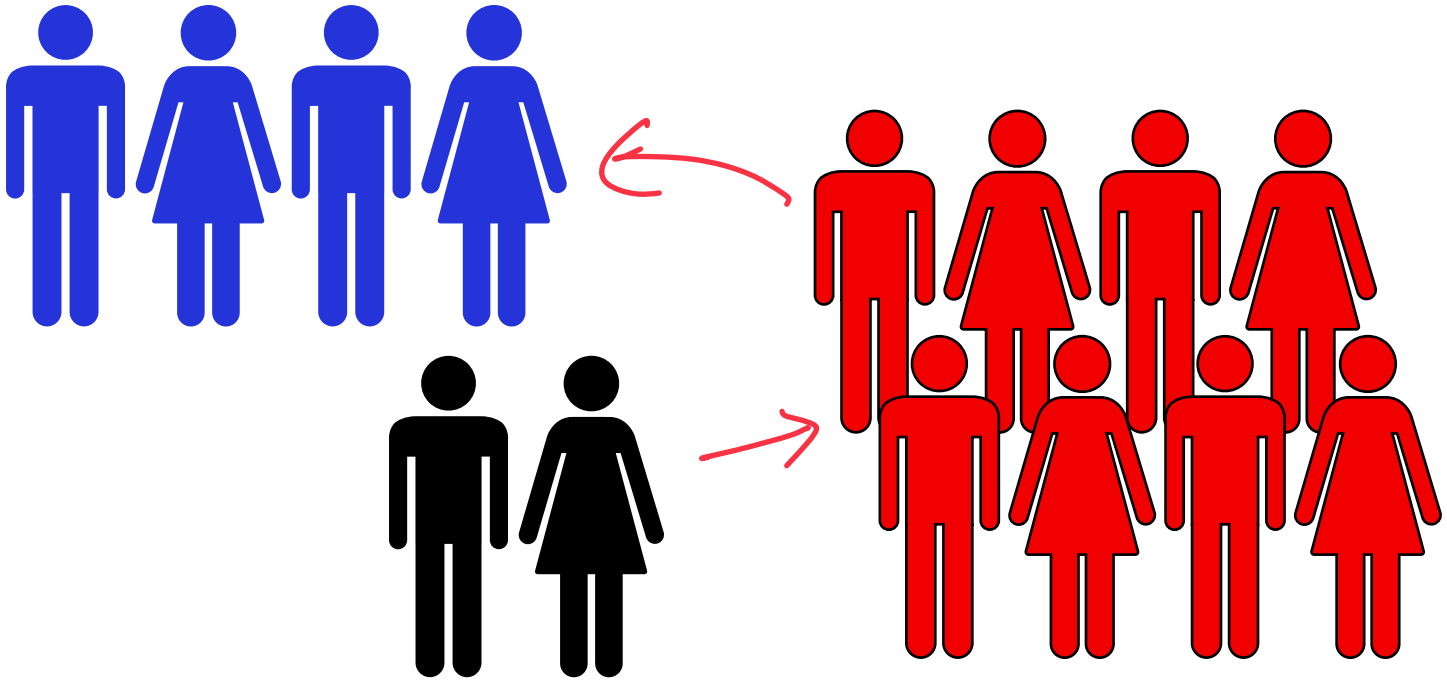
Discuss efficiency of disjoint sets

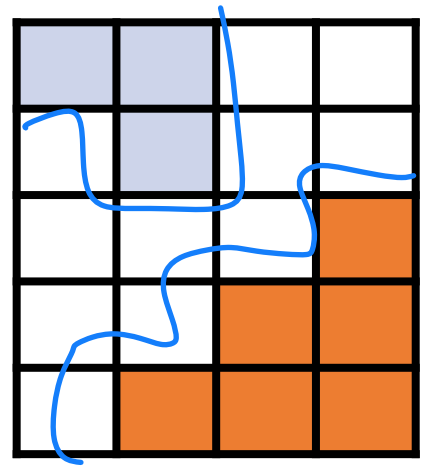
Identify improvements to implementation (and efficiency)

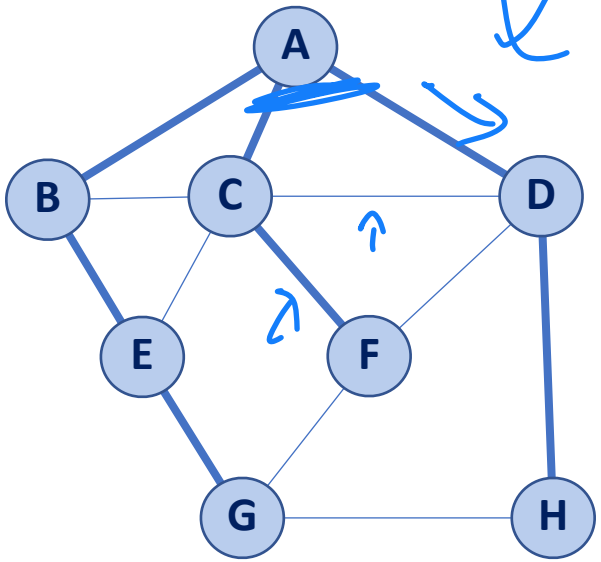
Storing and manipulating dynamic groups

We need a data structure which can efficiently look up (and change) group dynamics

Multiple datasets

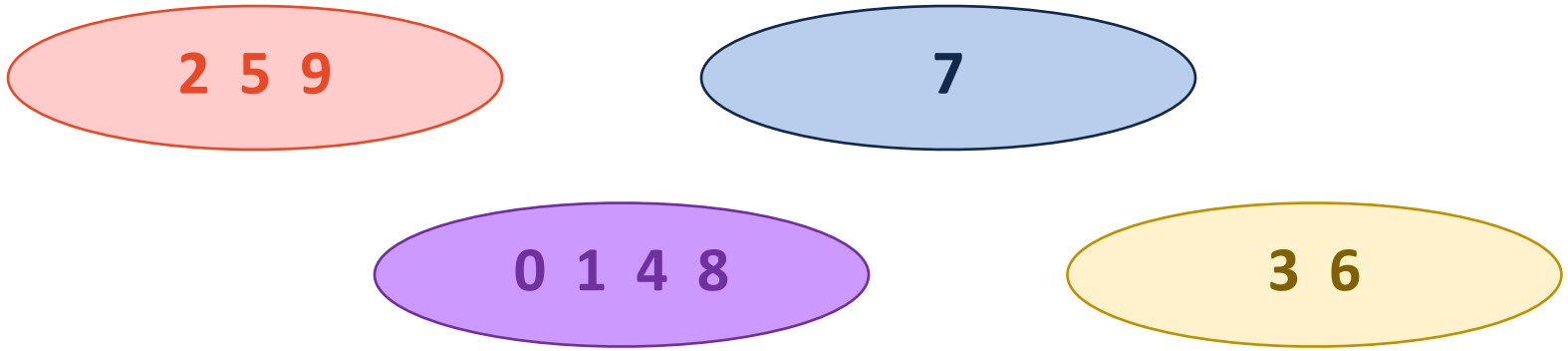






Disjoint Set ADT

A data structure designed to store relationships between items



Operations:

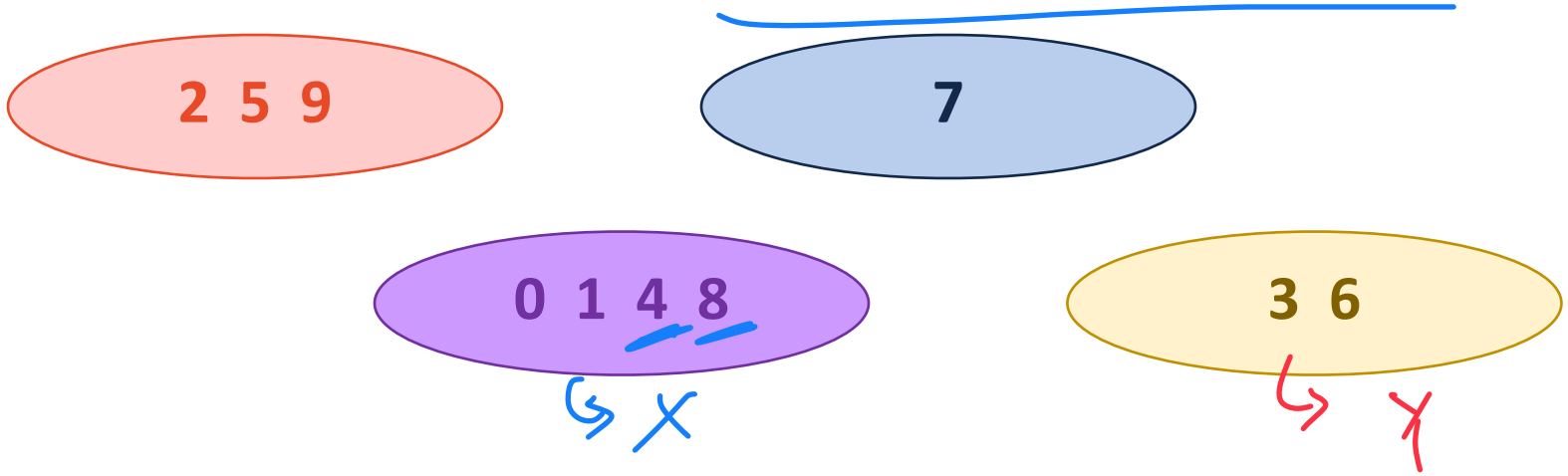
`find(k)` — returns "set representation" for item x

`union(s1, s2)` — Merge s1 and s2 into one set

`Constructor` — Make a new set

Disjoint Sets ‘Set Representation’

All items in a set have the same ‘Set Representation’



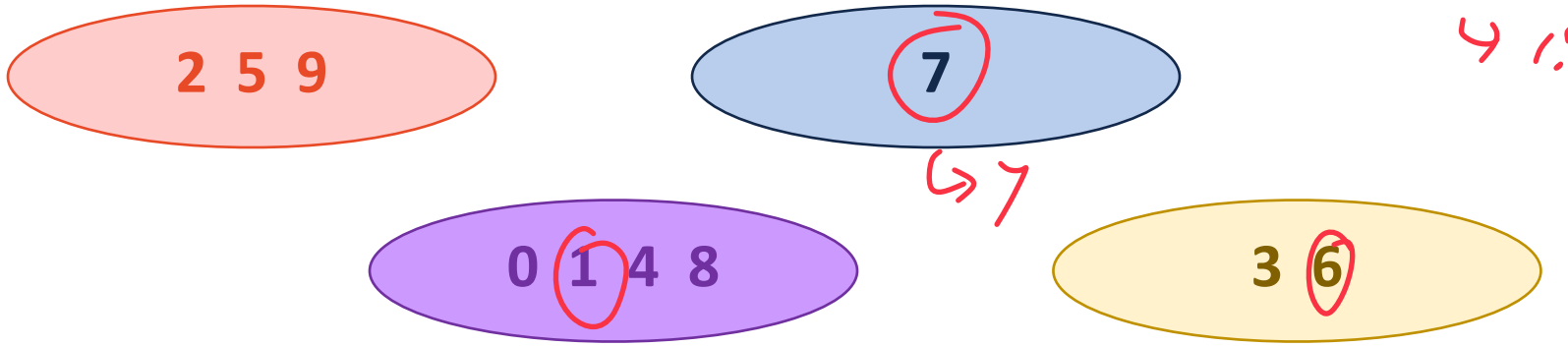
Operation:

```
find(4) == find(8)

find(4) != find(3)
```

Disjoint Sets 'Set Representation'

Each set is represented by a **canonical element** (internally defined)



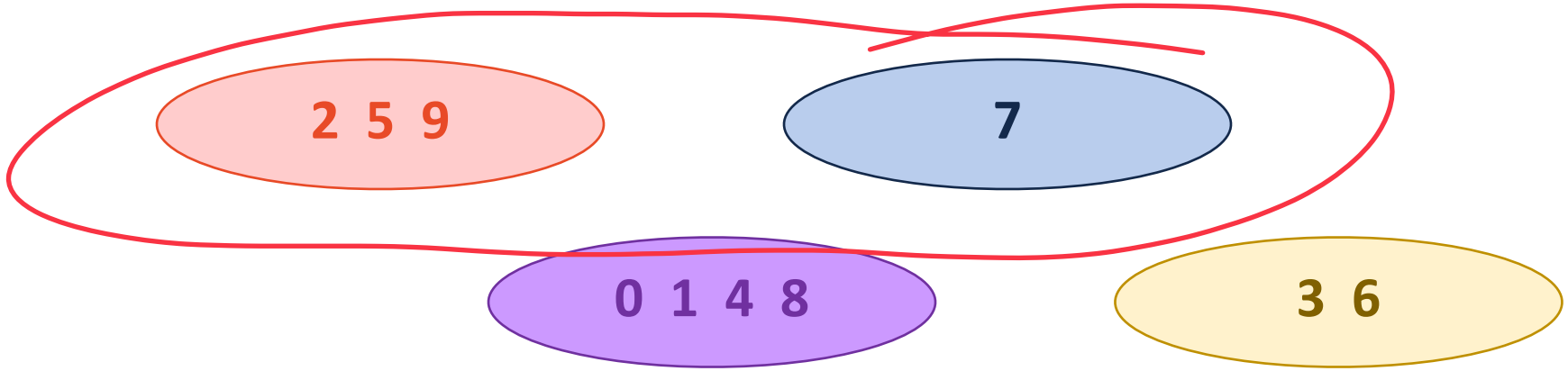
Operation:

```
find(4) == find(8)
```

```
find(4) != find(3)
```

Disjoint Sets

The union operation combines two sets into one set.



Operation:

```
if find(2) != find(7){
   union( 2, 7 );
}
```

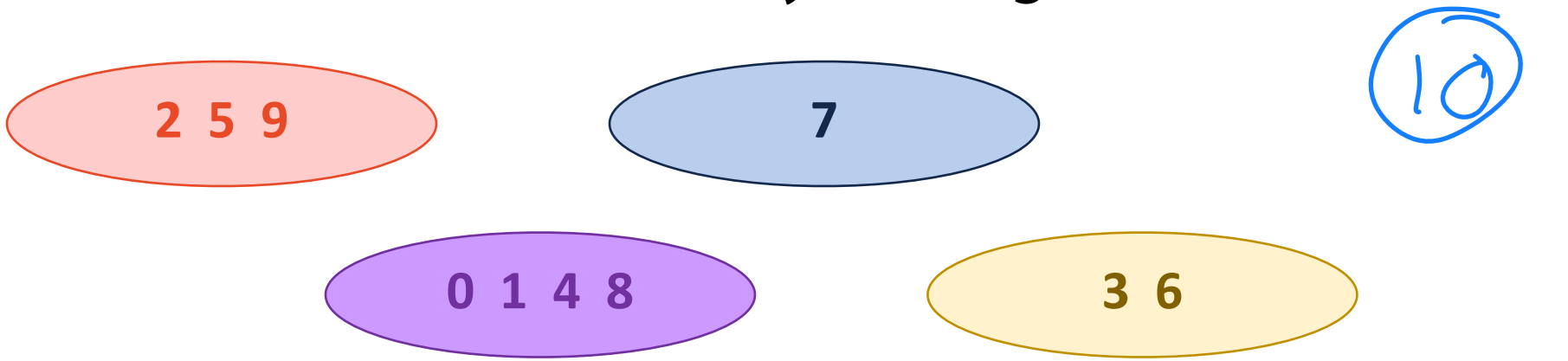
Does 7 join 259?

259 join 7?

implicitly we expect not to be in same set

Disjoint Sets

We add new items to our 'universe' by making new sets.



Operation:

```
makeSet(10);
```

↳ if want to add to set to union operation

Disjoint Sets

ADT:

```
makeSet(vector<T> items)
Find(T key)
Union(T k1, T k2)
```

Key Ideas:

1) Every item exists in exactly one set

2) Every item in each set has same representation

↓ Both imply

Every set has a different representation

Disjoint Sets

How might we implement a disjoint set?

↳ Store each set as a list

↳ Find? Array find O(n)

↳ union (insert smaller list into larger) O(m)

$|X| = n \quad |Y| = m$

$n > m$

↳ Dictionary | Map

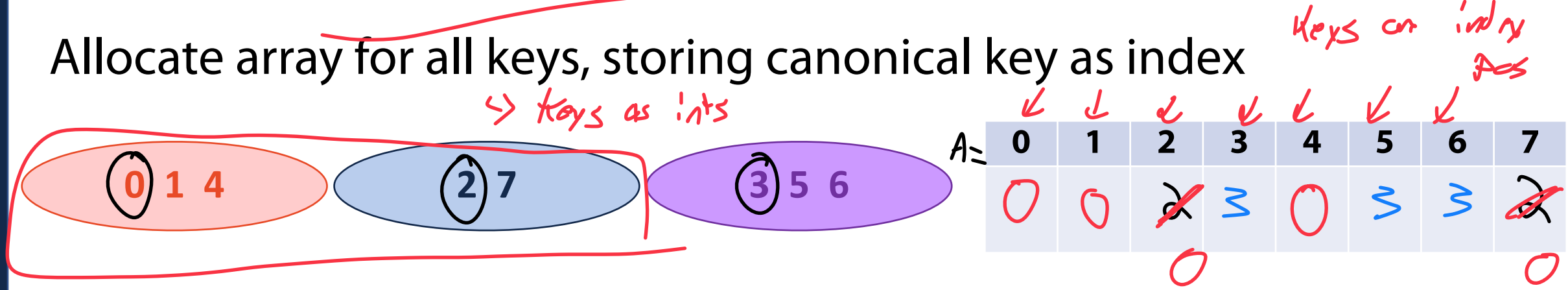
↳ Find() O(1)* as hash table

O(log n)

↳ union?

Implementation #1

Allocate array for all keys, storing canonical key as index



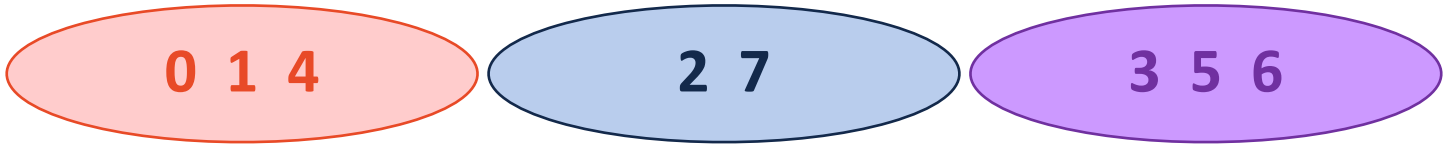
Find(k): Look up A[k] O(1) :)

Union(k_1, k_2): Walk across entire array and modify canonical items

O(n) :(

Implementation #1

Allocate array for all keys, storing canonical key as index



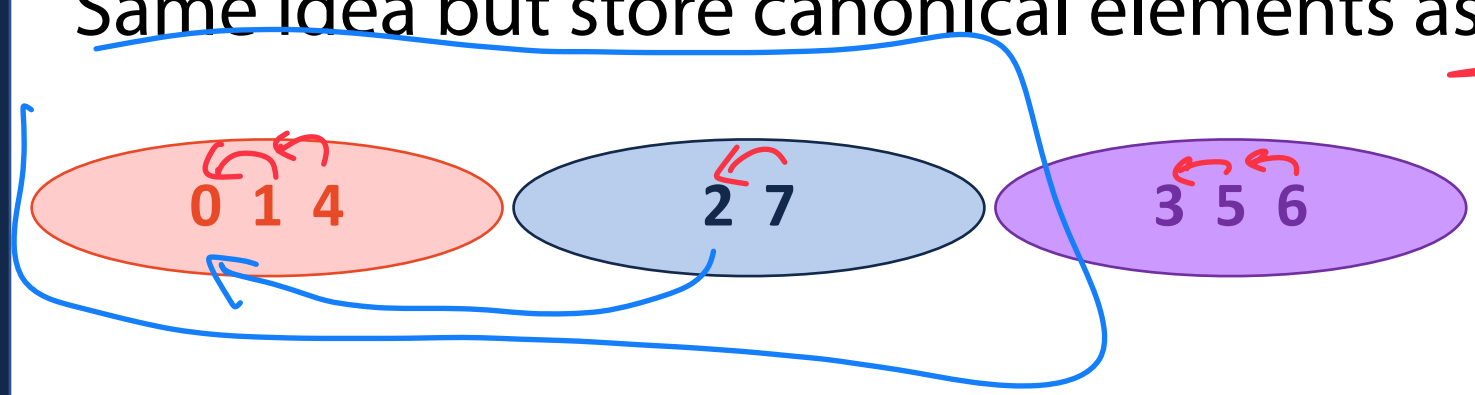
0	1	2	3	4	5	6	7
4	4	7	5	4	5	5	7

Find(k): Look up value in array

Union(k_1, k_2): Update **every item** in one set with new representation

Implementation #2

Same idea but store canonical elements as **-1**



0	1	2	3	4	5	6	7
-1	0	-1	-1	0	3	3	2

Find(k): Walk up chain to find canonical item $O(n)$

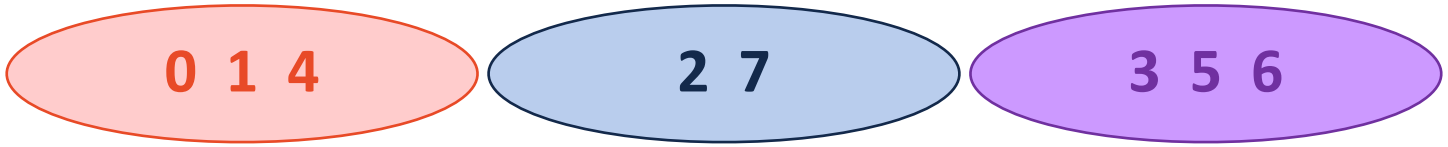
Union(k_1, k_2): Change one value to be the canonical item $O(1)$

* canonical items!!!

is in this example 2 has value -1

Implementation #2

Same idea but store canonical elements as **-1**



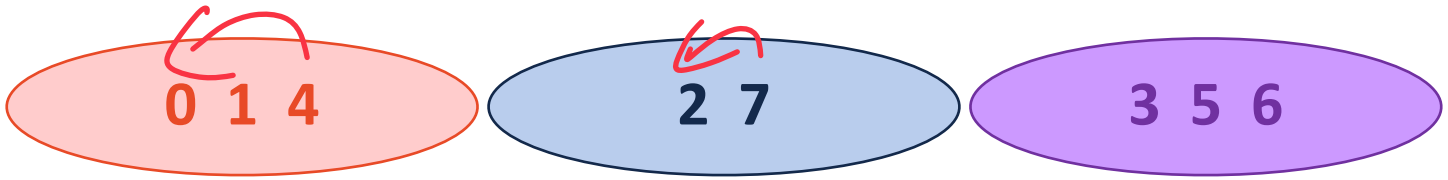
0	1	2	3	4	5	6	7
-1	0	-1	-1	0	3	3	2

Find(k): Repeatedly look up values until **-1**

Union(k_1, k_2): Update one canonical item to point at the other

Implementation #2

Same idea but store canonical elements as **-1**



0	1	2	3	4	5	6	7
-1	0	-1	-1	0	3	3	2

`Union(4, 7):`

Step 1: Get canonical item

2 finds per union

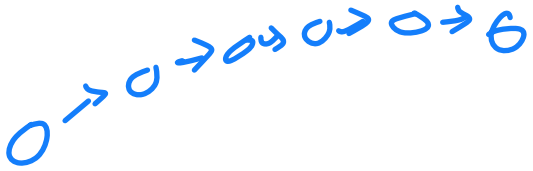
if you don't give me canonical!

Step 2: Easy! change value

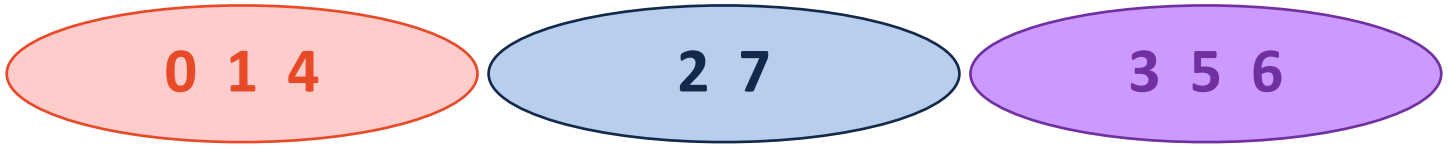
O(1) !!

`Find(7):`

Implementation #2



Same idea but store canonical elements as **-1**



0	1	2	3	4	5	6	7
-1	0	**0**	-1	0	3	3	2

`Union(4, 7):`

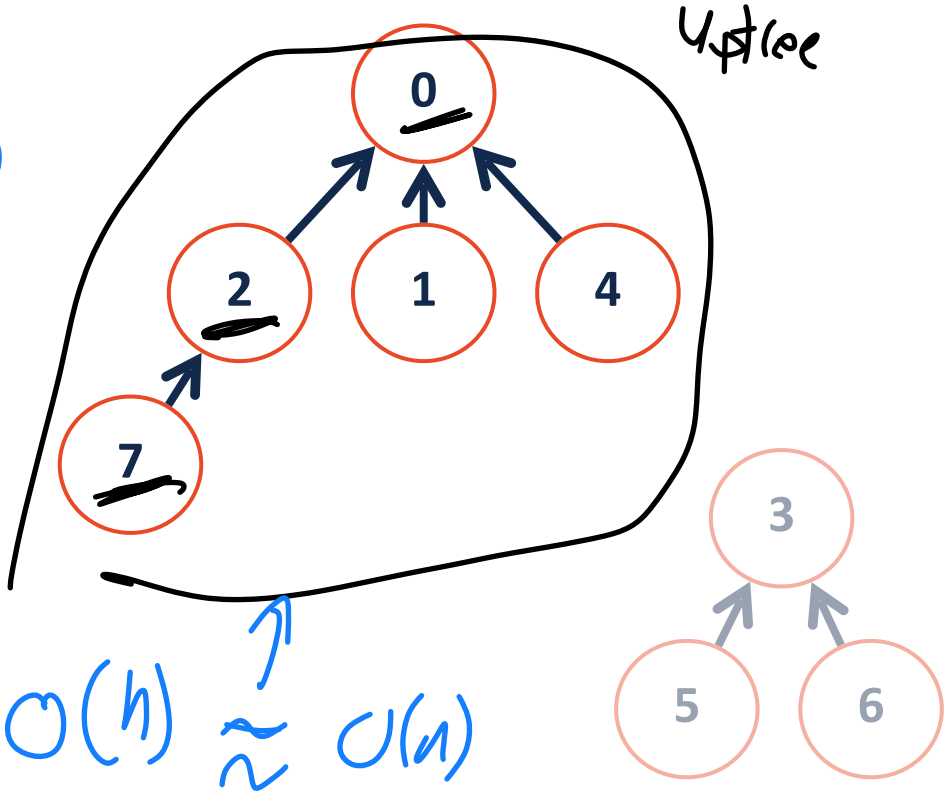
1) Get canonical element for 4 and 7 O(h)

`A[4] is 0` `A[7] is 2`

2) Have one point to the other O(1)

`Set` `A[2]=0`

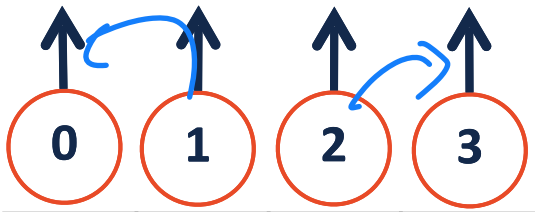
`Find(7):` `A[7] -> A[2] -> A[0]` O(h) ≈ O(n)



UpTrees

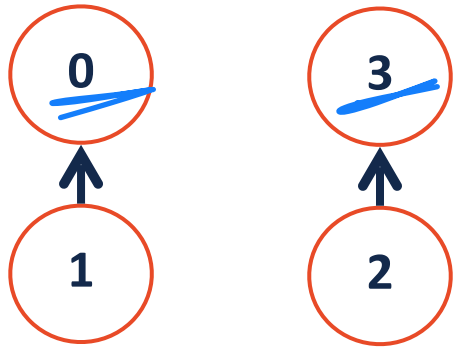
Join Code: 225

A



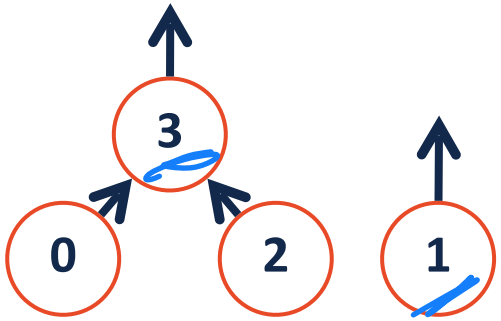
0	1	2	3
-1	-1	-1	-1

B



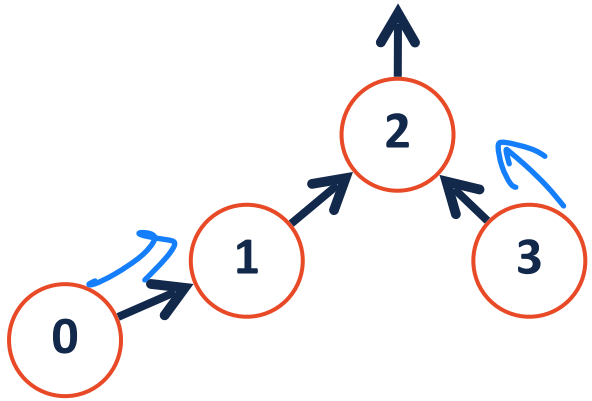
0	1	2	3
-1	0	3	-1

C



0	1	2	3
3	-1	3	-1

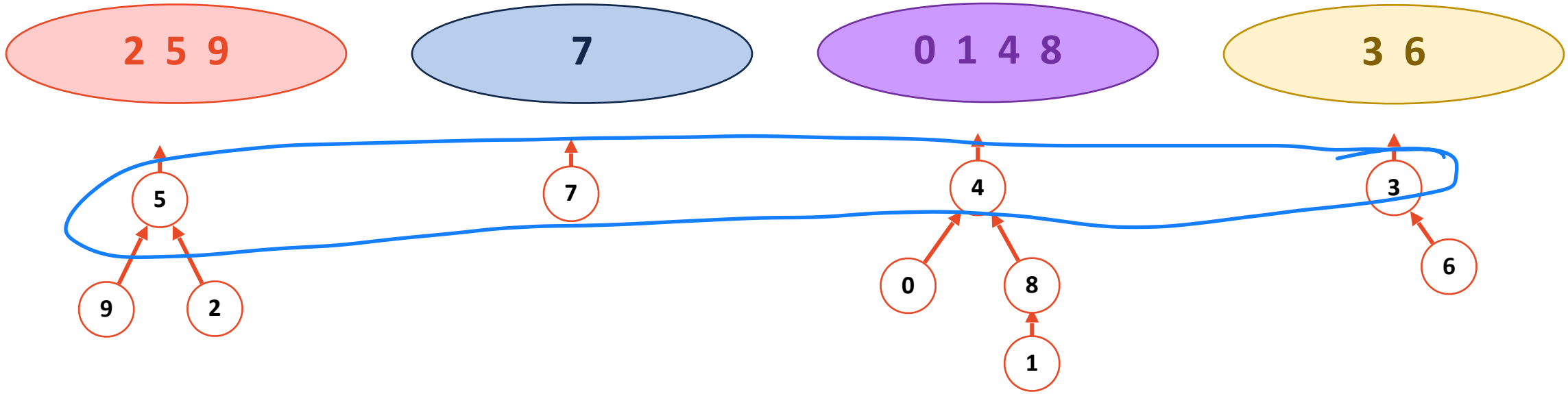
D



0	1	2	3
1	2	-1	2

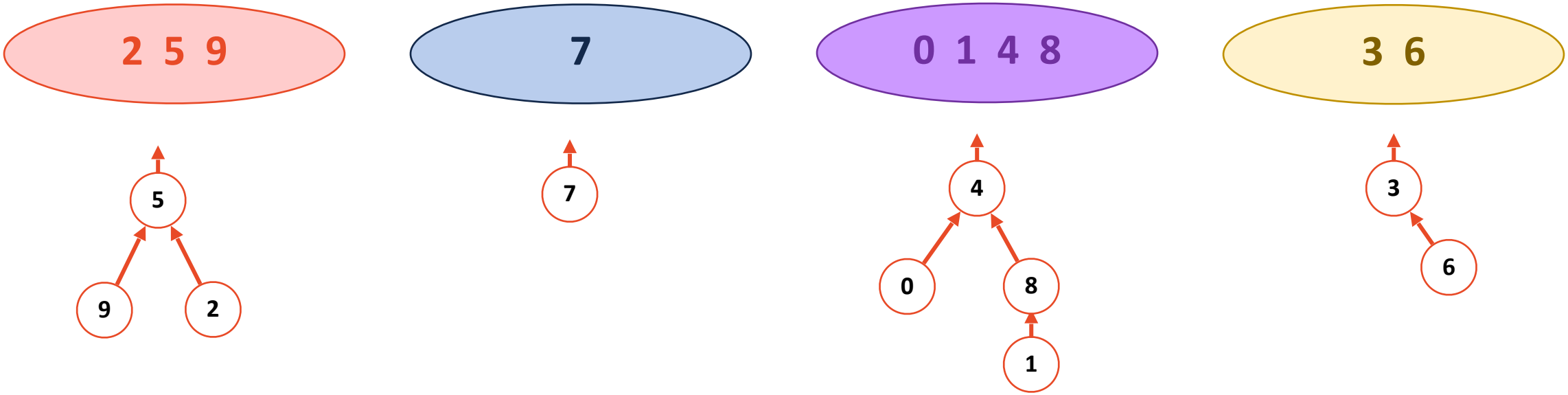
Disjoint Sets

Join Code: 225



0	1	2	3	4	5	6	7	8	9
4	8	5	-1	-1	-4	3	-4	4	5

Disjoint Sets



0	1	2	3	4	5	6	7	8	9
4	8	**5**	-1	-1	-1	3	-1	4	5

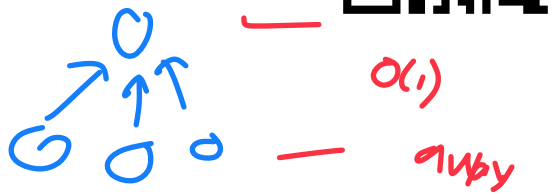
UpTrees Best and Worst Case

Join Code: 225

What does a best case UpTree look like?

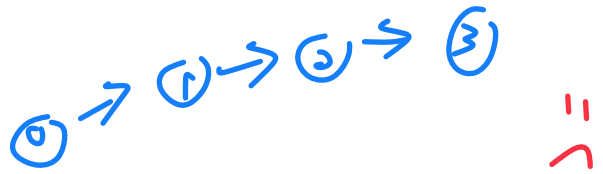
Both optimal!

A flat tree is best! O(h) is our bound



0	1	2	3
-1	-1	-1	-1
2	-1	2	2

What does a worst case UpTree look like?



0	1	2	3
1	2	3	-1

Disjoint Sets Representation

Implemented as an array where the value of key is index in array

The values inside the array stores our sets as an **UpTree**

The value **-1** is our representative element (the root)

All other set members store the index to a parent of the UpTree

Big O for Find: O(h)

Big O for Union: O(h) if not given canonical

O(1) if given canonical

Make a table

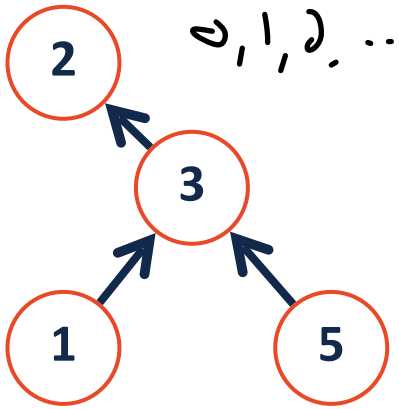
A = 0

B = 1

(=)

D = 3

if values at 0, 1, 2, ...



Disjoint Sets Find

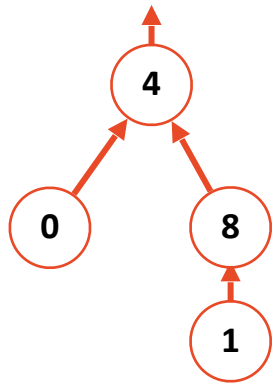
Find(1)

```
1  int DisjointSets::find(int i) {
2    if ( s[i] < 0 ) { return i; }
3    else { return find( s[i] ); }
4  }
```

canonical item

0 1 4 8

recursive find call!



Running time?

O(h)

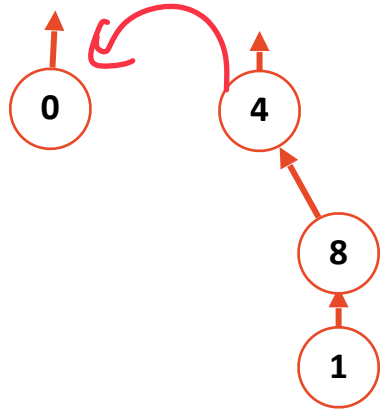
What is ideal UpTree?

0	1	2	3	4	5	6	7	8	9
4	8			-1				4	

Disjoint Sets Union

Union(0, 4)

```
1  int DisjointSets::union(int r1, int r2) {
2    // Naive Implementation assumes canonical
3
4    s[r2] = r1;
5  }
```

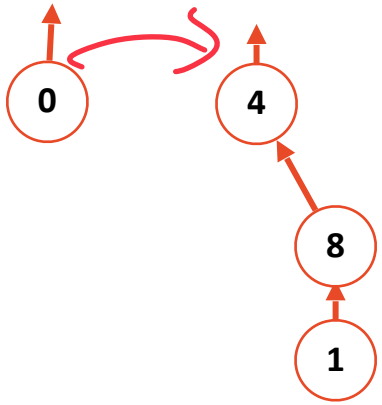


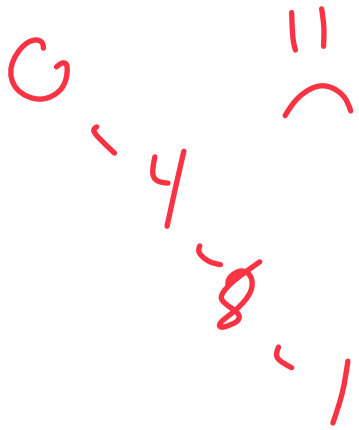
0	1	2	3	4	5	6	7	8	9
-1	8			-1				4	

Disjoint Sets Union

Union(4, 0)

```
1  int DisjointSets::union(int r1, int r2) {
2    // Naive Implementation assumes canonical
3
4    s[r2] = r1;
5  }
```





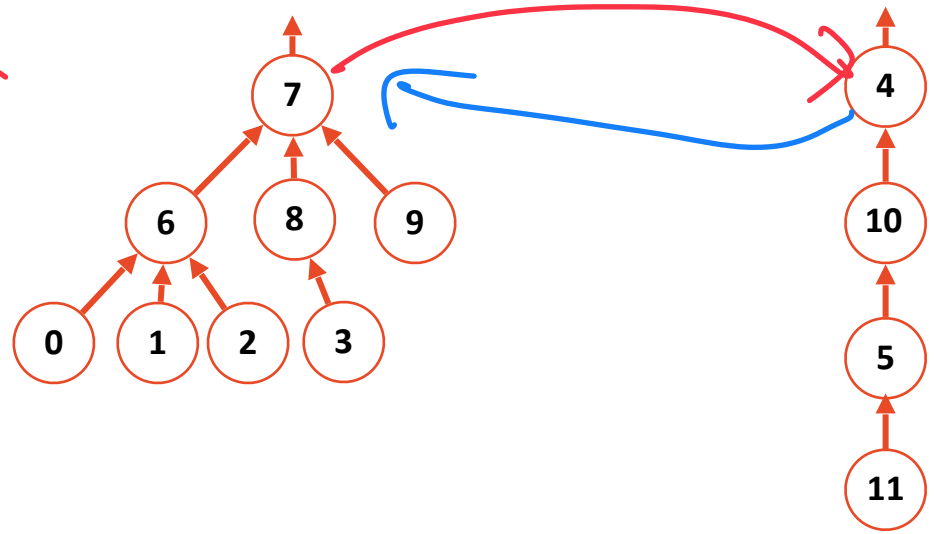
0	1	2	3	4	5	6	7	8	9
~~1~~	8			-1				4	

Disjoint Sets – Union

How do I want to merge these sets?

Height 2

8 items total



Height 3

4 items total

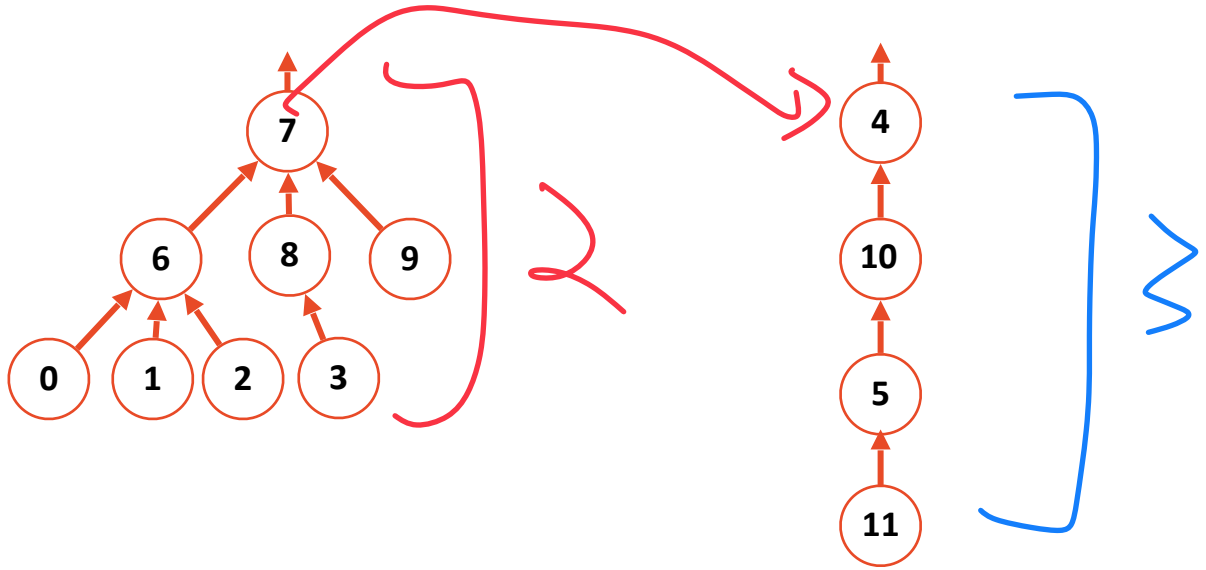
0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-1	10	7	-1	7	7	4	5

```
Union(4, 7)

Union(7, 4)
```

Join Code: 225

Disjoint Sets – Smart Union



Union by height

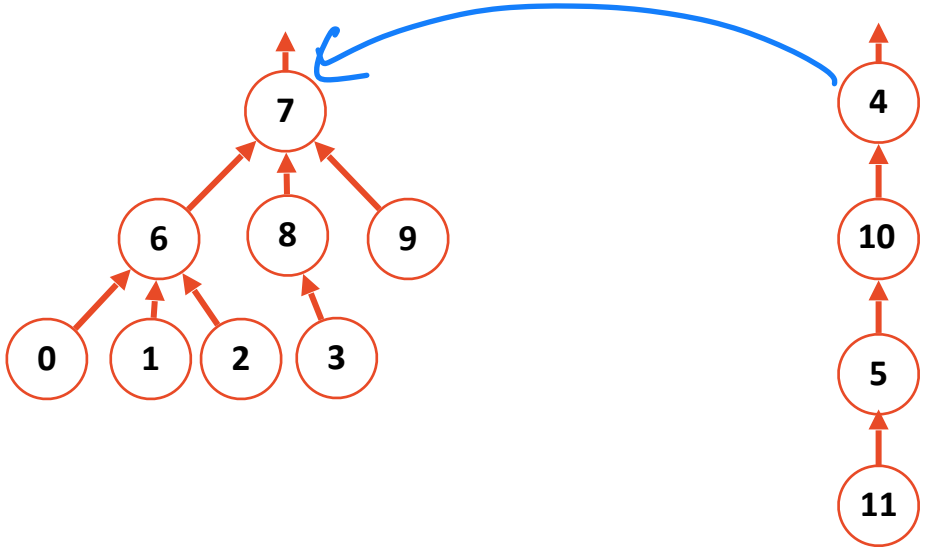
0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-4	10	7	-3	7	7	4	5

Idea: *Keep the height of the tree as small as possible.*

Clever Trick: If we union by height, store -1*(height+1) in canonical!

Disjoint Sets – Smart Union

Union(7, 4)



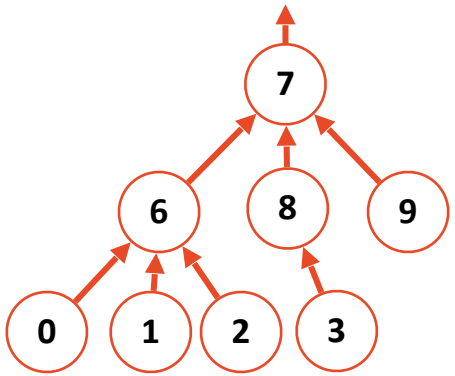
Union by size

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-4	10	7	-8	7	7	4	5

***Idea**: Minimize the number of nodes that increase in height*

Clever Trick: If we union by size, store -1*(size) in canonical!

Disjoint Sets – Smart Union



Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-4	10	7	4	7	7	4	5

Idea: *Keep the height of the tree as small as possible.*

Union by size

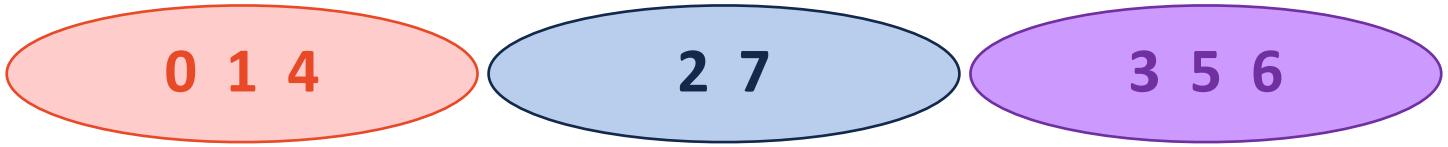
0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	7	10	7	-12	7	7	4	5

Idea: *Minimize the number of nodes that increase in height*

Both guarantee the height of the tree is: $O(\log n)$.

Disjoint Set Implementation

Store an UpTree as an array, canonical items store **height / size**



0	1	2	3	4	5	6	7
	0			0	3	3	2

Find(k): Repeatedly look up values until **negative value**

$O(\log n) \rightarrow O(1)^*$ "

Union(k_1, k_2): Update ***smaller*** canonical item to point to larger

Update value of remaining canonical item

$O(\log n) \rightarrow O(1)^*$ "